

Unix Programming 셸 프로젝트 #3

12161567 박기수

1. 제출일: 2020. 11. 27

2. 요구사항 정의

- 1) 'cd' 명령이 제대로 먹히지 않는 버그를 수정
- 2) 'exit' 명령을 구현
- 3) 백그라운드 실행을 구현
- 4) SIGCHLD로 자식 프로세스 wait() 시 프로세스가 온전하게 수행되도록 구현
- 5) ^C(SIGINT), ^W(SIGQUIT) 사용시 셸이 종료되지 않도록, Foreground 프로세스 실행 시 SIGINT를 받으면 프로세스가 끝나는 것을 구현
- 6) Redirection 구현
- 7) Pipe 구현

3. 구현 방법

1) 'cd' 명령이 제대로 먹히지 않는 버그를 수정

본 프로그램에서는 명령어를 입력하면 자식 프로세스를 생성해서 명령어를 처리한다. 하지만 `chdir()`의 경우에는 현재 프로세스의 위치만 변경하고 부모 프로세스의 위치는 변경할 수 없으므로 `cd` 명령 실행 후 `pwd`를 실행하면 현재 디렉토리의 위치가 변하지 않는다. 따라서, 자식 프로세스는 `chdir()`를 호출하지 않고 부모 프로세스에게 `signal`을 전달한다. `signal`을 전달받은 부모 프로세스는 `cd` 명령을 수행하는 시그널 핸들러를 호출하여 현재 디렉토리를 변경한다.

```
myshell> pwd
/home/ubuntu/unix/project
myshell> mkdir dirA
myshell> cd dirA
myshell> pwd
/home/ubuntu/unix/project/dirA
myshell> █
```

그림 1. cd 구현

2) 'exit' 명령을 구현

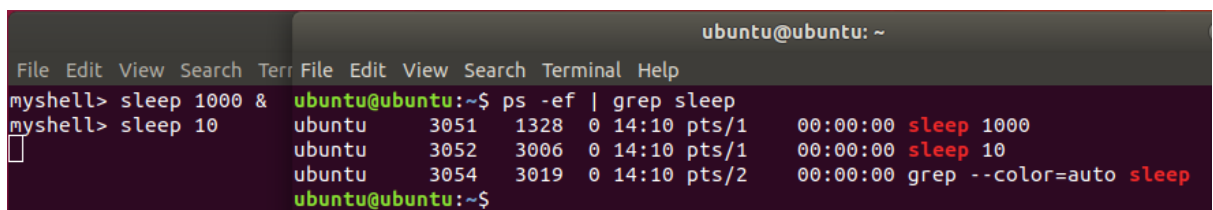
exit 명령어를 입력하면 자식 프로세스는 exit(-1)를 호출한다. 부모 프로세스에서는 wait(&status) 함수를 호출해서 자식 프로세스의 종료 상태를 받는다. 만약 WEXITSTATUS(status)의 값이 255이면 (-1을 전달하면 255로 전달됨) 부모 프로세스는 exit(0)을 호출해서 종료한다.

```
ubuntu@ubuntu:~/unix/project$ ./shell
myshell> exit
ubuntu@ubuntu:~/unix/project$
```

그림 2. exit 구현

3) 백그라운드 실행을 구현

백그라운드 명령이 호출되면 자식 프로세스는 fork()를 호출해서 손자 프로세스를 생성한다. 손자 프로세스는 execvp()를 호출해서 입력 받은 명령어를 처리하는 프로세스로 변경 후 해당 명령어를 처리한다. fork()를 호출한 자식 프로세스는 wait()를 호출하지 않고 즉시 종료 후 부모 프로세스로 복귀한다. 따라서, 손자 프로세스는 백그라운드에서 명령어를 처리하고, 부모 프로세스는 myshell> 로 복귀해서 다른 명령을 처리할 수 있다.



```
ubuntu@ubuntu: ~
File Edit View Search Terr File Edit View Search Terminal Help
myshell> sleep 1000 &
myshell> sleep 10
ubuntu@ubuntu:~$ ps -ef | grep sleep
ubuntu      3051    1328  0 14:10 pts/1    00:00:00 sleep 1000
ubuntu      3052    3006  0 14:10 pts/1    00:00:00 sleep 10
ubuntu      3054    3019  0 14:10 pts/2    00:00:00 grep --color=auto sleep
ubuntu@ubuntu:~$
```

그림 3. 백그라운드 구현

● 좀비 프로세스가 생기는 이유

자식 프로세스는 손자 프로세스를 생성한 후 wait()를 호출하지 않고 즉시 종료한다. 손자 프로세스는 자신의 부모(자식 프로세스)가 먼저 종료했으므로 고아 프로세스가 된다. 고아 프로세스는 init 프로세스에게 입양되는데, init 프로세스는 wait()를 자주 호출하지 않는다. 따라서, init에게 호출된 손자 프로세스는 실행이 종료된 후 init 프로세스에서 wait()를 호출하기 전까지 좀비 프로세스 상태가 된다.

● 백그라운드 기다리는 테스트 문제

백그라운드로 sleep을 실행하고 ps 명령을 실행하면 프로세스 목록에 sleep이 출력된다. myshell을 종료 후 재실행 하고 ps 명령어를 입력해도 종료 전에 실행했던 sleep이 출력된다.

```

ubuntu@ubuntu:~$ cd unix/project/
ubuntu@ubuntu:~/unix/project$ ./shell
myshell> sleep 10 &
myshell> sleep 20 &
myshell> ps
  PID TTY          TIME CMD
  5385 pts/2        00:00:00 bash
  5397 pts/2        00:00:00 shell
  5411 pts/2        00:00:00 sleep
  5413 pts/2        00:00:00 sleep
  5414 pts/2        00:00:00 ps
myshell> exit
ubuntu@ubuntu:~/unix/project$ ./shell
myshell> ps
  PID TTY          TIME CMD
  5385 pts/2        00:00:00 bash
  5411 pts/2        00:00:00 sleep
  5413 pts/2        00:00:00 sleep
  5416 pts/2        00:00:00 shell
  5417 pts/2        00:00:00 ps
myshell>

```

그림 4. 종료 전에 실행했던 프로세스가 그대로 출력됨

4) SIGCHLD로 자식 프로세스 wait() 시 프로세스가 온전하게 수행되도록 구현

일반적으로 system call은 작업이 완료될 때까지 시그널의 효과가 없다. 하지만, slow system call인 wait()의 경우에는 signal을 받으면 인터럽트 된다. 사용자 프로세스에서 인터럽트가 되면 시그널이 발생했던 위치로 정확하게 복귀하지만, slow system call은 시그널이 발생했던 위치로 복귀할 수 없고 에러를 반환한다. 따라서, 자식 프로세스가 종료됐을 경우 wait()를 정상적으로 실행이 안될 수도 있다. 이를 방지하기 위해서 SIGCHLD를 받으면 시그널 핸들러 내부에서 waitpid()를 호출해서 좀비 프로세스 생성을 방지한다.

5) ^C(SIGINT), ^W(SIGQUIT) 사용시 셸이 종료되지 않도록, Foreground 프로세스 실행 시 SIGINT를 받으면 프로세스가 끝나는 것을 구현

^C(SIGINT), ^W(SIGQUIT) 사용 시 셸이 종료되지 않도록 하는 가장 기본적인 방법은 SIG_IGN을 호출해서 해당 시그널을 무시하는 것이다. 하지만 SIGINT를 받을 때마다 새 프롬프트를 출력하는 기존 셸과 달리, SIGINT를 받아도 새 프롬프트가 출력되지 않는 문제점이 발생한다. 이를 해결하기 위해서 sigsetjmp()와 siglongjmp()를 사용했다. 프롬프트를 출력하는 코드 바로 앞에 sigsetjmp()를 설정하고 SIGINT와 SIGQUIT을 받을 경우 시그널 핸들러에서 siglongjmp()를 호출하여 프롬프트를 출력하는 부분으로 이동한다. 이렇게 구현할 경우 실제 셸처럼 ^C가 입력될 때마다 프롬프트가 출력된다.

```
myshell> ^C^C^C^C^C
```

```

myshell> ^C
myshell> ^C
myshell> ^C
myshell> ^C
myshell>

```

그림 5. SIG_IGN 사용 시 프롬프트 출력 안됨

그림 6. siglongjmp 사용 시 프롬프트 정상 출력됨

또한, Foreground 프로세스 실행 시에는 SIGINT를 받으면 종료되어야 하므로 fork()로 생성된 자식 프로세스는 SIGINT의 핸들러를 기본 핸들러(SIG_DFL)로 변경한다. Background 프로세스는 SIGINT를 받아도 계속 실행되므로, SIGINT의 핸들러를 무시하도록(SIG_IGN) 변경한다.

The image shows two terminal windows side-by-side. The left window is a standard Ubuntu terminal. The user runs 'sleep 1000 &' three times, which returns process IDs [1] 2945, [2] 2946, and [3] 2947. Then, the user presses Ctrl-C (^C), which immediately terminates the foreground process. Finally, the user runs 'ps', showing a list of processes including the three sleeping background processes (PIDs 2935, 2945, 2946, 2947, 2949) and the current shell process (PID 2937). The right window is a 'myshell' prompt. The user also runs 'sleep 1000 &' three times. After pressing Ctrl-C (^C), the 'myshell' prompt does not terminate but instead shows the 'ps' command output, which is identical to the one in the left window, demonstrating that the background processes continue to run despite the foreground signal.

그림 7. SIGINT 받으면 Foreground 프로세스 종료됨.

6) Redirection 구현

입력을 변경하는 redirection_read() 함수와 출력을 변경하는 redirection_write() 함수 두 가지로 구현했다. 두 함수는 명령어가 저장된 char** cmdvector와 명령어의 개수가 저장된 int* arg_cnt를 인자로 받는다. redirection을 처리하는 함수는 redirection 기호(<, >)를 찾는 과정, 입출력을 변경하는 과정, 명령어를 재구성하는 과정 총 3단계로 진행된다.

첫 번째는 기호를 찾는 과정이다. for문을 이용해서 cmdvector를 탐색하다가 redirection 기호와 일치하면 for문을 빠져나온다. for문을 빠져나왔을 때 i == arg_cnt이면 cmdvector 내에 redirection 기호가 없다는 의미이므로 함수를 종료한다. i == arg_cnt - 1이면 cmdvector의 제일 마지막이 redirection 기호라는 의미인데, redirection 기호 뒤에는 반드시 입, 출력으로 지정한 파일명이 와야 하므로 문법 오류를 발생시키고 종료한다.

두 번째는 입출력을 변경하는 과정이다. redirection_read() 함수는 dup2()를 이용해서 표준 입력을 의미하는 0번 file descriptor를 입력파일의 file descriptor로 변경하고 입력파일의 file descriptor를 닫는다. redirection_write() 함수는 dup2()를 이용해서 표준 출력을 의미하는 1번 file descriptor를 출력파일의 file descriptor로 변경하고 출력파일의 file descriptor를 닫는다.

세 번째는 명령어를 재구성하는 과정이다. redirection_read() 함수는 cmdvector에서 redirection 기호가 있던 위치부터 명령어를 앞으로 한 칸씩 이동시키고, 가장 마지막 명령어가 있던 위치를 NULL로 변경한다. redirection_write() 함수는 cmdvector에서 redirection 기호가 있던 위치부터 명령어를 앞으로 두 칸씩 이동시키고, 마지막 명령어와 이전 명령어의 위치를 NULL로 변경한다.

	0	1	2	3	4	5
변경 전	cat	<	hi.txt	>	hi2.txt	&
변경 후	cat	hi.txt	>	hi2.txt	&	NULL

그림 8. redirection_read() 실행 결과

	0	1	2	3	4
변경 전	cat	hi.txt	>	hi2.txt	&
변경 후	cat	hi.txt	&	NULL	NULL

그림 9. redirection_write() 실행 결과

```

myshell> cat > hi.txt
hi
hello
bye
myshell> cat < hi.txt
hi
hello
bye
myshell> cat < hi.txt > hi2.txt
myshell> cat hi2.txt
hi
hello
bye
myshell>

```

그림 10. redirection 구현

7) Pipe 구현

명령어를 입력하면 chk_pipe() 함수를 호출해서 파이프가 있는지 확인하고, 파이프가 있으면 파이프를 처리하는 execute_pipe() 함수를 실행한다. execute_pipe() 함수는 명령어를 나누는 과정, 명령어를 처리하는 과정 총 2단계로 진행된다. 명령어를 나눌 때에는 파이프 기호를 기준으로 나눈다.

cat < ls.txt | grep ^d | wc -l > dir_num.txt

1. 명령어 나누기

pipe_cnt: 2

pipevector	[0]	[1]	[2]	[3]
[0]	cat	<	ls.txt	
[1]	grep	^d		
[2]	wc	-l	>	dir_num.txt

그림 11. 파이프 명령어 나누기

명령어를 처리하는 과정은 파이프 개수만큼 fork()를 실행해서 처리한다. 자식 프로세스는 dup2()를 이용해서 표준 출력을 파이프로 변경하고 명령어를 실행한다. 부모 프로세스는 표준 입력을 파이프로 변경한다. 만약 fork()를 pipe_cnt 만큼 실행했으면 부모 프로세스에서 마지막 명령어를 처리한다. 명령어에 백그라운드 기호(&)가 포함 돼있으면 명령어를 백그라운드로 실행한다.

2. pipe_cnt만큼 fork() 실행

i = 0

자식	부모
p[0][0] p[0][1] (출력) cat < ls.txt	p[0][0] (입력) p[0][1]

i = 1

자식	부모
p[1][0] p[1][1] (출력) grep ^d	p[1][0] (입력) p[1][1] wc -l > dir_num.txt

그림 12. 파이프 처리 과정

```
myshell> ls -l > ls.txt &
myshell> cat ls.txt
total 48
drwxrwxr-x 2 ubuntu ubuntu 4096 Oct 21 20:53 dir1
drwxrwxr-x 2 ubuntu ubuntu 4096 Oct 31 11:41 dirA
drwxrwxr-x 2 ubuntu ubuntu 4096 Nov 25 19:10 dynamic
-rw-r--r-- 1 ubuntu ubuntu 0 Nov 27 01:15 ls.txt
-rwxrwxr-x 1 ubuntu ubuntu 18152 Nov 27 01:14 shell
-rw-rw-r-- 1 ubuntu ubuntu 9638 Nov 27 01:14 shell.c
drwxrwxr-x 2 ubuntu ubuntu 4096 Nov 25 19:11 testdir
myshell> cat < ls.txt | grep ^d | wc -l > dir_num.txt &
myshell> cat dir_num.txt
4
myshell> █
```

그림 13. 파이프 구현 (백그라운드 정상 작동)

4. Pseudo-code

1) cd 구현

● 시그널 핸들러

```
static void sig_chdir(int 시그널 번호) {  
    if 시그널 번호 == SIGUSR1 then  
        if chdir(경로) == -1 then  
            오류 출력;  
        }  
}
```

● 시그널 등록

```
signal(SIGUSR1, sig_chdir);
```

● main 함수

```
switch( pid = fork() ) {  
    case 0: // 자식 프로세스면  
        ...  
        if 명령어 == "cd" then  
            부모 프로세스에게 시그널 전달;  
            exit(0);  
        ...  
}
```

2) exit 구현

● 자식 프로세스

```
switch( pid = fork() ) {  
    case 0: // 자식 프로세스면  
        ...  
        if 명령어 == "exit" then  
            exit (-1);  
        ...  
}
```

● 부모 프로세스

```
switch( pid = fork() ) {  
    ...  
    default: // 부모 프로세스면  
        wait(&status);  
        if 자식 프로세스 종료 then  
            if 종료 상태 == -1 then  
                exit(0);  
        }  
}
```

3) 백그라운드 구현

```
switch( pid = fork() ) {  
    case 0: // 자식 프로세스면  
        ...  
        if 명령어에 & 포함 then  
            cmd := & 제외한 새 문자열;  
            pid = fork();  
  
            if 손자 프로세스 then  
                execvp(cmd);  
  
            else if 자식 프로세스 then  
                exit(0);  
            ...  
        }  
}
```

4) SIGCHLD 핸들러

```
static void sig_chld(int 시그널){
    if 시그널 == SIGCHLD then
        waitpid(-1, NULL, WNOHANG);
    }
}
```

5) SIGINT 핸들러, Foreground 프로세스 종료

● 처음으로 돌아가기

```
static void sig_goto_start(int 시그널) {
    if 시그널 == SIGINT || 시그널 == SIGQUIT then
        printf("\n");
        siglongjmp(jmpbuf, 1);
    }
}
```

● Foreground 프로세스

```
switch(pid=fork()){
    case 0:
        signal(SIGINT, SIG_DFL);
        signal(SIGQUIT, SIG_DFL);
```

● Background 프로세스

```
else if 명령어에 & 포함 then
    signal(SIGINT, SIG_IGN);
    signal(SIGQUIT, SIG_IGN);
    ...
```

6) redirection 처리

● redirection_read()

```
for i in range (0, arg_cnt)
    cmdvector[i] == NULL          then return
    cmdvector[i] == "<"           then break

if i == arg_cnt                  then return
if i == arg_cnt - 1              then fatal("error")

fd = open(...)
dup2(fd, 0)
close(fd)
for j in range (i, arg_cnt)
    cmdvector[j] = cmdvector[j+1]

cmdvector[j] <- NULL
arg_cnt <- arg_cnt - 1;
```

● redirection_write()

```
for i in range (0, arg_cnt)
    cmdvector[i] == NULL          then return
    cmdvector[i] == ">"           then break

if i == arg_cnt                  then return
if i == arg_cnt - 1              then fatal("error")

fd = open(...)
dup2(fd, 1)
close(fd)
for j in range (i, arg_cnt)
    cmdvector[j] = cmdvector[j+2]

cmdvector[j], cmdvector[j+1] <- NULL
arg_cnt <- arg_cnt - 2;
```


7) 파이프 처리

● chk_pipe()

// 파이프가 명령어 제일 처음이나 끝에 오면 문법 오류

```
if cmdvector[0] == "|" || cmdvector[arg_cnt - 1] == "|"      then fatal("pipe")

for i in range(0, arg_cnt)
    if cmdvector[i] == "|"      then return i
return 0
```

● execute_pipe()

```
for i in range(0, pipe_cnt)
    pipe(p[i])
    switch(pid = fork())
        case -1:    fatal("execute_pipe()")
        case 0:
            redirection_read(pipevector[i], &pipevector_size_list[i]);
            dup2(p[i][1], 1);
            close(p[i][1]);
            close(p[i][0]);

            if( i > 0){
                dup2(p[i-1][0], 0);
                close(p[i-1][0]);
            }
            execvp(pipevector[i][0], pipevector[i]);
            fatal("execute_pipe() - fork() case 0");
        default:
            dup2(p[i][0], 0);
            close(p[i][0]);
            close(p[i][1]);
            if(i == pipe_cnt - 1){
                redirection_write(pipevector[pipe_cnt], &pipevector_size_list[pipe_cnt])
                if 백그라운드 기호(&) 포함      then 백그라운드 처리
                execvp(pipevector[pipe_cnt][0], pipevector[pipe_cnt]);
                fatal("execute_pipe() - fork() default");
```